

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shivaraj Kallappa Pujari (1BM22CS259)

in partial fulfillment for the award of the degree of

**BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025**

**B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019**
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the Lab work entitled “ Bio Inspired Systems (23CS5BSBIS)” carried out by **Shivaraj Kallappa Pujari (1BM22CS259)**, who is bonafide student of **B.M.S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE
---	---

Index

Sl. No.	Date	Experiment Title	Page No.
1	24-10-2024	Genetic Algorithm for Optimization Problems	1-5
2	07-11-2024	Particle Swarm Optimization for Function Optimization	5-11
3	14-11-2024	Ant Colony Optimization for the Traveling Salesman Problem	11-20
4	21-11-2024	Cuckoo Search (CS)	20-27
5	28-11-2024	Grey Wolf Optimizer (GWO)	27-35
6	18-12-2024	Parallel Cellular Algorithms and Programs	36-42
7	18-12-2024	Optimization via Gene Expression Algorithms	43-49

Github Link:

https://github.com/Shivarajpujari2004/BIS_LAB-1BM22CS259

Program 1: Genetic Algorithm for Optimization on Problems

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:

24/10/23

Genetic Algorithm



Page _____

18 Genetic Algorithm:-

* Pseudocode:-

1. Initialize population of size 'pop-size' with random values in range $[x\text{-range-low}, x\text{-range-high}]$

2. For generation in range (generations):

a. Evaluate fitness of each individual:

for each individual x_i in population:
fitness(x_i) = $x_i \times x_i$

b. Select 2 parents based on fitness value (roulette-wheel selection):

total_fitness = sum of all fitness values

for each individual x_i in population:

probability(x_i) = fitness(x_i) / total_fitness

Select 2 parents (p_1, p_2) based on probability

c. perform crossover with probability 'crossover_rate':

If random-number < crossover_rate:

alpha = random value between 0 and 1

offspring1 = alpha * $p_1 + (1 - \alpha) * p_2$

offspring2 = alpha * $p_2 + (1 - \alpha) * p_1$

Else:

offspring1 = p_1

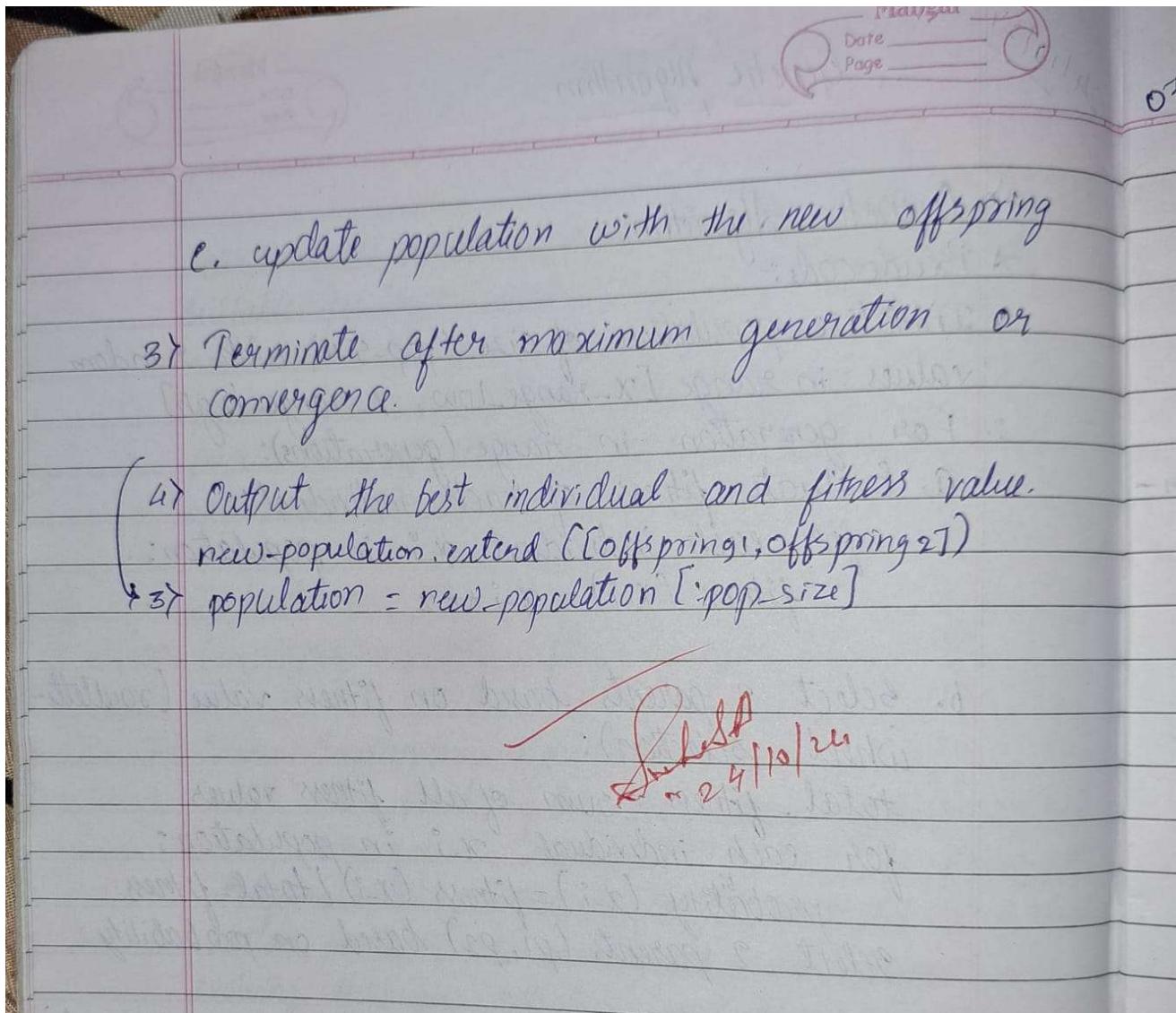
offspring2 = p_2

d. Apply mutation with probability 'mutation_rate':

For each offspring:

If random-number < mutation_rate:

offspring = random value in range $[x\text{-range-low}, x\text{-range-high}]$



Code:

```

#lab-2: genetic
import numpy as np
import random

# Objective function to maximize
def objective_function(x):
    return x ** 2

# Initialize parameters
population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7
range_min = -10
  
```

```

range_max = 10

# Create initial population
def initialize_population(size, min_val, max_val):
    return np.random.uniform(min_val, max_val, size)

# Evaluate fitness of the population
def evaluate_fitness(population):
    return np.array([objective_function(x) for x in population])

# Selection using roulette-wheel method
def selection(population, fitness):
    total_fitness = np.sum(fitness)
    probabilities = fitness / total_fitness
    return population[np.random.choice(range(len(population)), size=2, p=probabilities)]

# Crossover between two parents
def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        return (parent1 + parent2) / 2 # Simple averaging for crossover
    return parent1 # No crossover

# Mutation of an individual
def mutate(individual):
    if random.random() < mutation_rate:
        return np.random.uniform(range_min, range_max)
    return individual

# Genetic Algorithm function
def genetic_algorithm():
    # Step 1: Initialize population
    population = initialize_population(population_size, range_min, range_max)

    for generation in range(num_generations):
        # Step 2: Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        best_index = np.argmax(fitness)
        best_solution = population[best_index]
        best_fitness = fitness[best_index]

        # print(f'Generation {generation + 1}: Best Solution = {best_solution}, Fitness = {best_fitness}')

        # Step 3: Create new population
        new_population = []

```

```

for _ in range(population_size):
    # Select parents
    parent1, parent2 = selection(population, fitness)
    # Crossover to create offspring
    offspring = crossover(parent1, parent2)
    # Mutate offspring
    offspring = mutate(offspring)
    new_population.append(offspring)

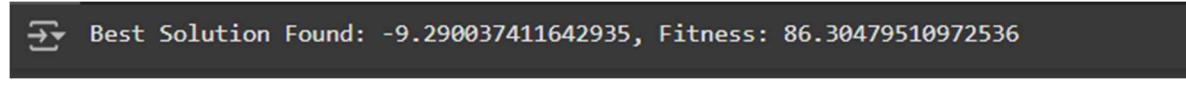
# Step 6: Replace old population with new population
population = np.array(new_population)

return best_solution, best_fitness

# Run the Genetic Algorithm
best_solution, best_fitness = genetic_algorithm()
print(f"Best Solution Found: {best_solution}, Fitness: {best_fitness}")

```

OUTPUT:



```

Best Solution Found: -9.290037411642935, Fitness: 86.30479510972536

```

Program 2: Particle Swarm Optimization for Function Optimization

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of particles, inertia weight, cognitive and social coefficients.
3. Initialize Particles: Generate an initial population of particles with random positions and velocities.
4. Evaluate Fitness: Evaluate the fitness of each particle based on the optimization function.
5. Update Velocities and Positions: Update the velocity and position of each particle based on its own best position and the global best position.
6. Iterate: Repeat the evaluation, updating, and position adjustment for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

07/11/24

PSO

Mangal

Date _____
Page _____

2) Particle Swarm Optimization (PSO):

a) Rastrigin objective function:

→ This function calculates the Rastrigin value based on particle's current position x . The goal of PSO is to minimize this function, so the global best solution will correspond to the minimum value of the Rastrigin function, which is 0 at the origin.

⇒ def function rastrigin(x):

$A = 10$

Initialize $res = Ax \cdot \text{len}(x)$

For each x_i in x :

$$res += x_i^2 - A * \cos(2\pi \cdot x_i)$$

Return res

2) Particle initialization:

→ Each particle starts with a random position and velocity. The initial best position is the current position.

⇒ Class particle:

Position = Random in [lower_bound, upper_bound]

velocity = Random in [-1, 1]

Best_position = Position

Best_value = Rastrigin(Position)

3) Velocity update rule:

→ Update the particle's velocity based on inertia, personal best and global best.

\Rightarrow Function update-velocity (particle, global-best position, w, c₁, c₂):

$$\gamma_1 = \text{Random}(0, 1)$$

$$\gamma_2 = \text{Random}(0, 1)$$

$$\text{Inertia} = w * \text{Particle.velocity}$$

$$\text{Cognitive} = c_1 * \gamma_1 * (\text{Particle.Best-position} - \text{Particle.Position})$$

$$\text{Social} = c_2 * \gamma_2 * (\text{Global-best-position} - \text{Particle.position})$$

$$\text{Particle.velocity} = \text{Inertia} + \text{Cognitive} + \text{Social}$$

\hookrightarrow Position Update Rule

\rightarrow Update the particle's position by adding the velocity, and ensure it stays within bounds.

\Rightarrow Function update-position (Particle, lower-bound, upper-bound):

$$\text{Particle.Position} += \text{Particle.Velocity}$$

$$\text{Particle.Position} = \text{clip}(\text{Particle.Position}, \text{lower-bound}, \text{upper-bound})$$

\hookrightarrow Fitness Evaluation:

\rightarrow Calculates Fitness and if it's better than the current best, update the particle's personal best.

\Rightarrow Function evaluate-fitness (Particle):

$$\text{Fitness} = \text{Rastrigin}(\text{Particle.position})$$

If Fitness < Particle.Best-value:

$$\text{Particle.Best-value} = \text{Fitness}$$

$$\text{Particle.Best-position} = \text{Particle.Position}$$

6) Main PSO Algorithm:

→ Iterate over particles, update their velocity, position and fitness. Track the global best solution.

⇒ Function Particle_Swarm_Optimization(dim, lower-bound, upper-bound, num-particles, max-iter, w, c₁, c₂):

 Initialize particles

 Set G_{global-best-position} = particles[0].Best-position.

 Set G_{global-best-value} = particles[0].Best-value

 For i=1 to max-iter:

 For each particle:

 update-velocity(particle, G_{global-best-position}, w, c₁, c₂)

 update-position(particle, lower-bound, upper-bound)

 Evaluate_Fitness(particle)

 If Particle.Best-value < G_{global-best-value}:

 G_{global-best-value} = particle.best-value

 G_{global-best-position} = particle.best-position

 P Print G_{global-best-value} (optional)

Return G_{global-best-position}, G_{global-best-value}.

7) Set the parameters & run the PSO algorithm, then print the solution.

Code:

```
#lab-3: pso
import numpy as np
import random

# Define the optimization problem (Rastrigin Function)
def rastrigin(x):
    A = 10
    return A * len(x) + sum([(xi**2 - A * np.cos(2 * np.pi * xi)) for xi in x])

# Particle Swarm Optimization (PSO) implementation
class Particle:
    def __init__(self, dimension, lower_bound, upper_bound):
        # Initialize the particle position and velocity randomly
        self.position = np.random.uniform(lower_bound, upper_bound, dimension)
        self.velocity = np.random.uniform(-1, 1, dimension)
        self.best_position = np.copy(self.position)
        self.best_value = rastrigin(self.position)

    def update_velocity(self, global_best_position, w, c1, c2):
        # Update the velocity of the particle
        r1 = np.random.rand(len(self.position))
        r2 = np.random.rand(len(self.position))

        # Inertia term
        inertia = w * self.velocity

        # Cognitive term (individual best)
        cognitive = c1 * r1 * (self.best_position - self.position)

        # Social term (global best)
        social = c2 * r2 * (global_best_position - self.position)

        # Update velocity
        self.velocity = inertia + cognitive + social

    def update_position(self, lower_bound, upper_bound):
        # Update the position of the particle
        self.position = self.position + self.velocity

        # Ensure the particle stays within the bounds
        self.position = np.clip(self.position, lower_bound, upper_bound)

    def evaluate(self):
        # Evaluate the fitness of the particle
        fitness = rastrigin(self.position)
```

```

# Update the particle's best position if necessary
if fitness < self.best_value:
    self.best_value = fitness
    self.best_position = np.copy(self.position)

def particle_swarm_optimization(dim, lower_bound, upper_bound, num_particles=30, max_iter=100,
w=0.5, c1=1.5, c2=1.5):
    # Initialize particles
    particles = [Particle(dim, lower_bound, upper_bound) for _ in range(num_particles)]

    # Initialize the global best position and value
    global_best_position = particles[0].best_position
    global_best_value = particles[0].best_value

    for i in range(max_iter):
        # Update each particle
        for particle in particles:
            particle.update_velocity(global_best_position, w, c1, c2)
            particle.update_position(lower_bound, upper_bound)
            particle.evaluate()

        # Update global best position if needed
        if particle.best_value < global_best_value:
            global_best_value = particle.best_value
            global_best_position = np.copy(particle.best_position)

        # Optionally print the progress
        if (i+1) % 10 == 0:
            print(f"Iteration {i+1}/{max_iter} - Best Fitness: {global_best_value}")

    return global_best_position, global_best_value

# Set the parameters for the PSO algorithm
dim = 2          # Number of dimensions for the function
lower_bound = -5.12  # Lower bound of the search space
upper_bound = 5.12   # Upper bound of the search space
num_particles = 30    # Number of particles in the swarm
max_iter = 100      # Number of iterations

# Run the PSO
best_position, best_value = particle_swarm_optimization(dim, lower_bound, upper_bound,
num_particles, max_iter)

# Output the best solution found
print("\nBest Solution Found:")
print("Position:", best_position)

```

```
print("Fitness:", best_value)
```

OUTPUT:

```
Iteration 10/100 - Best Fitness: 1.1103296669969005
Iteration 20/100 - Best Fitness: 0.020031338560627887
Iteration 30/100 - Best Fitness: 2.788695226740856e-06
Iteration 40/100 - Best Fitness: 1.0778596895022474e-06
Iteration 50/100 - Best Fitness: 6.450946443692374e-10
Iteration 60/100 - Best Fitness: 2.0463630789890885e-11
Iteration 70/100 - Best Fitness: 1.0658141036401503e-14
Iteration 80/100 - Best Fitness: 0.0
Iteration 90/100 - Best Fitness: 0.0
Iteration 100/100 - Best Fitness: 0.0

Best Solution Found:
Position: [-1.63024230e-09  1.14735681e-09]
Fitness: 0.0
```

Program 3: Ant Colony Optimization for the Traveling Salesman Problem

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.
2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.
3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.
4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.
5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.
6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:

→ 3) Ant Colony Optimization for Travelling Salesman Problem.
→ 4) calculate distances (cities)

→ This function calculates the Euclidean distances between all pairs of cities & stores them in a matrix.

⇒ function calculate_distances(cities);
 num_cities = length of cities

`num_cities = length of cities`

initialize an empty distance matrix with zeros
for each pair of cities (i, j) :

for each pair of cities (i, j) :

distance i,j = Euclidean distance between cities i and cities j

$$\text{distance}[j][i] = \text{distance}[i][j]$$

return distance matrix.

2) heuristic (distances)

→ This function computes the heuristic values (η) as the inverse of distances between cities. A higher value indicates a shorter distance E , therefore a more attractive path for the ants to follow.

\Rightarrow function heuristic (distances):

for each pair of cities (i, j) :

~~if distance[i][j] = 0
else cta[i][j] = 1 / distance[i][j]~~

else:

$$\text{eta}[i][j] = 0$$

return heuristic matrix (eta)

3) choose_next_city (pheromone, eta, visited)
→ The function calculates the probability of choosing each city based on pheromone trail & the heuristic (inverse of distance). The selection process is probabilistic, where ants are more likely to choose cities with stronger pheromone trails & shorter distances.

⇒ function choose_next_city (pheromone, eta, visited):
 initialize an empty list for probabilities
 for each city j :
 if city j is not visited:
 pheromone_{ij} = pheromone[visited[-1], j]
 raised to alpha
 heuristic_{ij} = eta[visited[-1], j] raised to beta
 probability = pheromone_{ij} * heuristic_{ij}
 add probability to the list

 normalize probabilities to make sum to 1
 return the city j with highest probability.

4) construct_solution (pheromone, eta)
→ Each ant starts at a random city and probabilistically constructs a tour by selecting the next city using the pheromone & heuristic information.

⇒ function construct_solution (pheromone, eta):
 initialize the tour with a random starting city
 while the tour is incomplete (i.e., not all cities are visited):
 select the next city using choose_next_city
 (pheromone, eta, tour)

add the city to the tour
return the complete tour.

5) update-pheromones (pheromone, all-tours, distances, best-tour):

→ The pheromone levels are updated in 2 steps:

a) Evaporation: The pheromone level on each edge decays over time

b) Deposition: Pheromone is deposited based on the inverse of tour lengths. Shorter tours receive more pheromone.

⇒ function update-pheromones (pheromone, all-tours, distances, best-tour):

$$\text{pheromone} \leftarrow (1 - \gamma) \text{pheromone}$$

for each tour in all-tours:

tour-length = compute total length of the tour using distances

for each city pair in the tour:

$$\text{pheromone}[\text{city1}, \text{city2}] \leftarrow 1 / \text{tour-length}$$

best-length = compute total length of best-tour

for each city pair in best-tour:

$$\text{pheromone}[\text{city1}, \text{city2}] \leftarrow 1 / \text{best-length}$$

6) run-aco (distances, num-iterations):

initialize pheromone matrix with initial pheromone value

initialize best-tour as None and best-length as infinity.

```

for iteration in range(1 to num_iterations):
    all_tours = []
    for each ant in range(num_ants):
        tour = construct_solution(pheromone, eta)
        add tour to all_tours
    all_lengths = compute the lengths of all
    tours using distances
    find the best tour in all_tours based on
    shortest length
    if the best tour in this iteration is shorter
    than the previous best:
        update best_tour and best_length
    return best_tour, best_length

```

→ The run_aro function simulates the process of multiple ants constructing solutions over several iterations.

7) plot_route(cities, best_tour)
 → The plot_route function generates a graphical representation of best tour found by ants.

→ function plot_route(cities, best_tour):
 create a new figure for plotting
 for each city in cities:
 plot the cities as a red dot and label it
 create a list of cities in the order of the best_tour
 append the first city at end to complete the tour loop
 plot the cities as a line connecting them in best_tour order
 add title and labels to the plot
 display plot.

Code:

```
#ant colony
import numpy as np
import matplotlib.pyplot as plt

# 1. Define the Problem: Create a set of cities with their coordinates
cities = np.array([
    [0, 0], # City 0
    [1, 5], # City 1
    [5, 1], # City 2
    [6, 4], # City 3
    [7, 8], # City 4
])
# Calculate the distance matrix between each pair of cities
def calculate_distances(cities):
    num_cities = len(cities)
    distances = np.zeros((num_cities, num_cities))

    for i in range(num_cities):
        for j in range(num_cities):
            distances[i][j] = np.linalg.norm(cities[i] - cities[j])

    return distances

distances = calculate_distances(cities)

# 2. Initialize Parameters
num_ants = 10
num_cities = len(cities)
alpha = 1.0 # Influence of pheromone
beta = 5.0 # Influence of heuristic (inverse distance)
rho = 0.5 # Evaporation rate
num_iterations = 30
initial_pheromone = 1.0

# Pheromone matrix initialization
pheromone = np.ones((num_cities, num_cities)) * initial_pheromone

# 3. Heuristic information (Inverse of distance)
def heuristic(distances):
    with np.errstate(divide='ignore'): # Ignore division by zero
        return 1 / distances

eta = heuristic(distances)
```

```

# 4. Choose next city probabilistically based on pheromone and heuristic info
def choose_next_city(pheromone, eta, visited):
    probs = []
    for j in range(num_cities):
        if j not in visited:
            pheromone_ij = pheromone[visited[-1], j] ** alpha
            heuristic_ij = eta[visited[-1], j] ** beta
            probs.append(pheromone_ij * heuristic_ij)
        else:
            probs.append(0)
    probs = np.array(probs)
    return np.random.choice(range(num_cities), p=probs / probs.sum())

# Construct solution for a single ant
def construct_solution(pheromone, eta):
    tour = [np.random.randint(0, num_cities)]
    while len(tour) < num_cities:
        next_city = choose_next_city(pheromone, eta, tour)
        tour.append(next_city)
    return tour

# 5. Update pheromones after all ants have constructed their tours
def update_pheromones(pheromone, all_tours, distances, best_tour):
    pheromone *= (1 - rho) # Evaporate pheromones

    # Add pheromones for each ant's tour
    for tour in all_tours:
        tour_length = sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)])
        for i in range(-1, num_cities - 1):
            pheromone[tour[i], tour[i + 1]] += 1.0 / tour_length

    # Increase pheromones on the best tour
    best_length = sum([distances[best_tour[i], best_tour[i + 1]] for i in range(-1, num_cities - 1)])
    for i in range(-1, num_cities - 1):
        pheromone[best_tour[i], best_tour[i + 1]] += 1.0 / best_length

# 6. Main ACO Loop: Iterate over multiple iterations to find the best solution
def run_aco(distances, num_iterations):
    pheromone = np.ones((num_cities, num_cities)) * initial_pheromone
    best_tour = None
    best_length = float('inf')

    for iteration in range(num_iterations):
        all_tours = [construct_solution(pheromone, eta) for _ in range(num_ants)]
        all_lengths = [sum([distances[tour[i], tour[i + 1]] for i in range(-1, num_cities - 1)]) for tour in all_tours]

```

```

current_best_length = min(all_lengths)
current_best_tour = all_tours[all_lengths.index(current_best_length)]

if current_best_length < best_length:
    best_length = current_best_length
    best_tour = current_best_tour

update_pheromones(pheromone, all_tours, distances, best_tour)

print(f"Iteration {iteration + 1}, Best Length: {best_length}")

return best_tour, best_length

# Run the ACO algorithm
best_tour, best_length = run_aco(distances, num_iterations)

# 7. Output the Best Solution
print(f"Best Tour: {best_tour}")
print(f"Best Tour Length: {best_length}")

# 8. Plot the Best Route
def plot_route(cities, best_tour):
    plt.figure(figsize=(8, 6))
    for i in range(len(cities)):
        plt.scatter(cities[i][0], cities[i][1], color='red')
        plt.text(cities[i][0], cities[i][1], f'City {i}', fontsize=12)

    # Plot the tour as lines connecting the cities
    tour_cities = np.array([cities[i] for i in best_tour] + [cities[best_tour[0]]]) # Complete the loop by
    returning to the start
    plt.plot(tour_cities[:, 0], tour_cities[:, 1], linestyle='-', marker='o', color='blue')

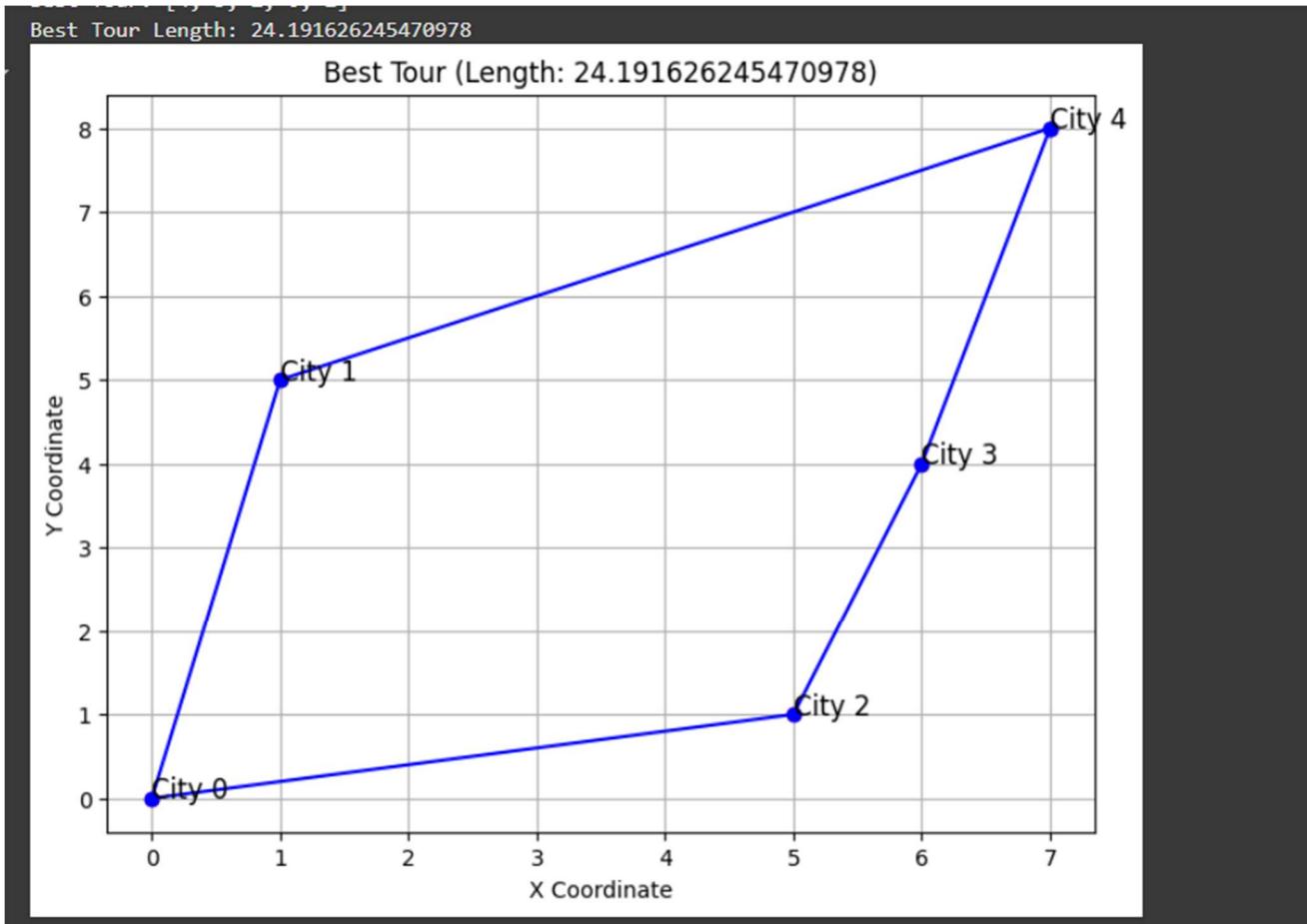
    plt.title(f"Best Tour (Length: {best_length})")
    plt.xlabel("X Coordinate")
    plt.ylabel("Y Coordinate")
    plt.grid(True)
    plt.show()

# Call the plot function
plot_route(cities, best_tour)

```

OUTPUT:

```
→ Iteration 1, Best Length: 24.191626245470978
Iteration 2, Best Length: 24.191626245470978
Iteration 3, Best Length: 24.191626245470978
Iteration 4, Best Length: 24.191626245470978
Iteration 5, Best Length: 24.191626245470978
Iteration 6, Best Length: 24.191626245470978
Iteration 7, Best Length: 24.191626245470978
Iteration 8, Best Length: 24.191626245470978
Iteration 9, Best Length: 24.191626245470978
Iteration 10, Best Length: 24.191626245470978
Iteration 11, Best Length: 24.191626245470978
Iteration 12, Best Length: 24.191626245470978
Iteration 13, Best Length: 24.191626245470978
Iteration 14, Best Length: 24.191626245470978
Iteration 15, Best Length: 24.191626245470978
Iteration 16, Best Length: 24.191626245470978
Iteration 17, Best Length: 24.191626245470978
Iteration 18, Best Length: 24.191626245470978
Iteration 19, Best Length: 24.191626245470978
Iteration 20, Best Length: 24.191626245470978
Iteration 21, Best Length: 24.191626245470978
Iteration 22, Best Length: 24.191626245470978
Iteration 23, Best Length: 24.191626245470978
Iteration 24, Best Length: 24.191626245470978
Iteration 25, Best Length: 24.191626245470978
Iteration 26, Best Length: 24.191626245470978
Iteration 27, Best Length: 24.191626245470978
Iteration 28, Best Length: 24.191626245470978
Iteration 29, Best Length: 24.191626245470978
Iteration 30, Best Length: 24.191626245470978
Best Tour: [4, 3, 2, 0, 1]
Best Tour Length: 24.191626245470978
```



Program 4: Cuckoo Search (CS)

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.
3. Initialize Population: Generate an initial population of nests with random positions.
4. Evaluate Fitness: Evaluate the fitness of each nest based on the optimization function.
5. Generate New Solutions: Create new solutions via Lévy flights.
6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.
7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.
8. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

21/4/24

Cuckoo - Search

Mangal
Date _____
Page _____

4) Cuckoo search Algorithm:

1) Objective Function:

→ It calculates the fitness of a solution by summing the squares of its elements. It represents the optimization problem, where lower values indicate better solutions.

⇒ Function $\text{obj}(x)$:

fitness = 0
for each element in x :
 fitness += element²
Return fitness.

2) Levy flight function:

→ This function generates a random step size based on a levy distribution, enabling exploration of search space with both small & large steps. It is crucial for avoiding local minima & encouraging global optimization.

⇒ Function $\text{levy_flight}(\lambda)$:

$$\sigma_u = \sqrt{\frac{(\Gamma(1+\lambda) - \sin(\pi\lambda/2))}{\Gamma((1+\lambda)/2)}} \cdot \lambda^{\lambda/2}$$

$$u = \text{Random Normal}(0, \sigma_u)$$

$$v = \text{Random Normal}(0, \sigma_v)$$

$$\text{step} = u / \text{abs}(v) \cdot (1/\lambda)$$

Return step.

3) Cuckoo Search Algorithm:-

→ This is the main optimization function that iteratively update solutions using levy flights and replacement strategies. It maintains a population of "nests", evaluates their fitness, abandons bad solutions, & tracks the best solution over iterations.

→ a) Initialize population: Initialize nests Randomly, calculate their fitness & identify best nest.

⇒ Function CS(~~num_nests, num_iterations, discovery_rate, dim, lower, upper~~):

~~nests = Randomuniform(lower, upper, size = (num_nests, dim))~~

~~fitness = [objective-fun(nest) for nest in nests]~~

~~best_nest_idx = Arg min(fitness)~~

~~best_nest = nests[best_nest_idx]~~

~~best_fitness = fitness[best_nest_idx]~~

~~lambda = 1.5~~

~~fitness_history = []~~

b) Main Optimization loop:

⇒ For it=1 to num_iterations:

For i=1 to num_nests-1:

~~step_size = levy-flight(lambda)~~

~~new_soln = nests[i] + step_size * (nests[i] - best_nest)~~

~~new_soln = clip(new_soln, lower, upper)~~

~~new_fitness = obj-function(new_soln)~~

~~If new_fitness < fitness[i]:~~

~~nests[i] = new_soln~~

~~fitness[i] = new_fitness.~~

c) Abandon Worst Nests:

$\Rightarrow \text{num_discover} = \text{Floor}(\text{discovery_rate} * \text{num_nests})$
 $\text{random_nests} = \text{Random choice}(\text{Range}(0, \text{num_nests}))$,
 $\text{num_discover, replace} = \text{False}$)

For nest_idx in random_nests:

$\text{nests}[\text{nest_idx}] = \text{RandomUniform}(\text{lower}, \text{upper}, d)$
 $\text{fitness}[\text{nest_idx}] = \text{obj_fun}(\text{nests}[\text{nest_idx}])$

d) Update Best nest:

$\Rightarrow \text{curr_best_idx} = \text{ArgMin}(\text{fitness})$

If $\text{fitness}[\text{curr_best_idx}] < \text{best_fitness}$:

$\text{best_nest} = \text{nests}[\text{curr_best_idx}]$

$\text{best_fitness} = \text{fitness}[\text{curr_best_idx}]$

e) Track fitness & print progress

$\Rightarrow \text{fitness_history}.append(\text{best_fitness})$

$\text{print("Iterations", iteration, "Best fitness", best_fitness)}$

f) Plot results & Return Best Solution

$\Rightarrow \text{Plot}(\text{fitness_history}, \text{title} = \text{"Fitness Convergence over Iterations"}, \text{xlabel} = \text{"Iteration"}, \text{ylabel} = \text{"Best Fitness"})$

Return best_nest, best_fitness

~~Abel SB
22/11/24~~

Code:

```
#cuckoo search
import numpy as np
import random
import math
import matplotlib.pyplot as plt

# Define a sample function to optimize (Sphere function in this case)
def objective_function(x):
    return np.sum(x ** 2)

# Lévy flight function
def levy_flight(Lambda):
    sigma_u = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, size=1)
    v = np.random.normal(0, sigma_v, size=1)
    step = u / (abs(v) ** (1 / Lambda))
    return step

# Cuckoo Search algorithm
def cuckoo_search(num_nests=25, num_iterations=100, discovery_rate=0.25, dim=5, lower_bound=-10, upper_bound=10):
    # Initialize nests
    nests = np.random.uniform(lower_bound, upper_bound, (num_nests, dim))
    fitness = np.array([objective_function(nest) for nest in nests])

    # Get the current best nest
    best_nest_idx = np.argmin(fitness)
    best_nest = nests[best_nest_idx].copy()
    best_fitness = fitness[best_nest_idx]

    Lambda = 1.5 # Parameter for Lévy flights
    fitness_history = [] # To track fitness at each iteration

    for iteration in range(num_iterations):
        # Generate new solutions via Lévy flight
        for i in range(num_nests):
            step_size = levy_flight(Lambda)
            new_solution = nests[i] + step_size * (nests[i] - best_nest)
            new_solution = np.clip(new_solution, lower_bound, upper_bound)
            new_fitness = objective_function(new_solution)

            # Replace nest if new solution is better
            if new_fitness < fitness[i]:
```

```

nests[i] = new_solution
fitness[i] = new_fitness

# Discover some nests with probability 'discovery_rate'
random_nests = np.random.choice(num_nests, int(discovery_rate * num_nests), replace=False)
for nest_idx in random_nests:
    nests[nest_idx] = np.random.uniform(lower_bound, upper_bound, dim)
    fitness[nest_idx] = objective_function(nests[nest_idx])

# Update the best nest
current_best_idx = np.argmin(fitness)
if fitness[current_best_idx] < best_fitness:
    best_fitness = fitness[current_best_idx]
    best_nest = nests[current_best_idx].copy()

# Store fitness for plotting
fitness_history.append(best_fitness)

# Print the best solution at each iteration (optional)
print(f"Iteration {iteration+1}/{num_iterations}, Best Fitness: {best_fitness}")

# Plot fitness convergence graph
plt.plot(fitness_history)
plt.title('Fitness Convergence Over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Best Fitness')
plt.show()

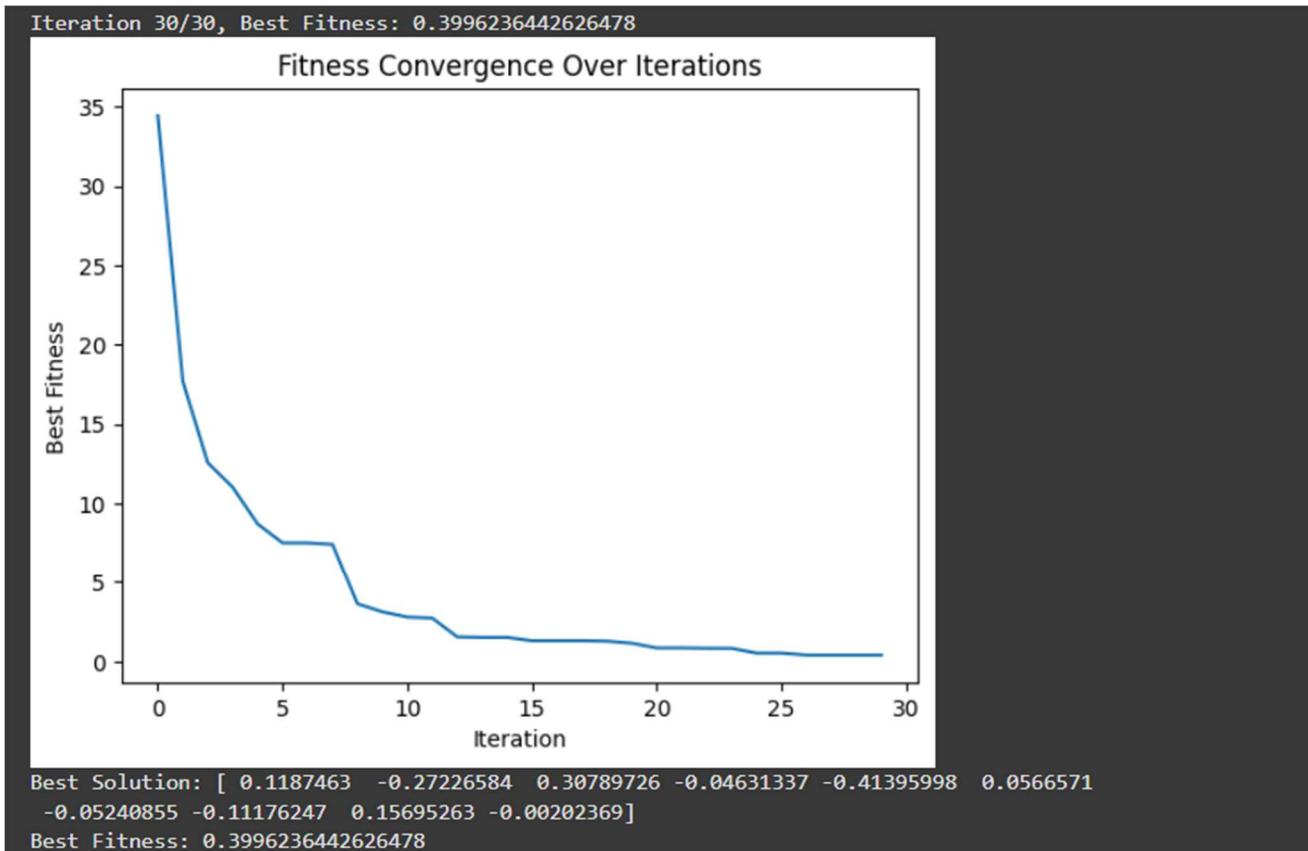
# Return the best solution found
return best_nest, best_fitness

# Example usage
best_nest, best_fitness = cuckoo_search(num_nests=30, num_iterations=30, dim=10, lower_bound=-5, upper_bound=5)
print("Best Solution:", best_nest)
print("Best Fitness:", best_fitness)

```

OUTPUT:

➡ Iteration 1/30, Best Fitness: 34.421347350368414
Iteration 2/30, Best Fitness: 17.701267864864427
Iteration 3/30, Best Fitness: 12.572246094152595
Iteration 4/30, Best Fitness: 11.025968548544025
Iteration 5/30, Best Fitness: 8.713786692960158
Iteration 6/30, Best Fitness: 7.5206125475077785
Iteration 7/30, Best Fitness: 7.5206125475077785
Iteration 8/30, Best Fitness: 7.426062303628502
Iteration 9/30, Best Fitness: 3.6305424687807872
Iteration 10/30, Best Fitness: 3.122312407680085
Iteration 11/30, Best Fitness: 2.7935374916676268
Iteration 12/30, Best Fitness: 2.7258275326189683
Iteration 13/30, Best Fitness: 1.5451154817432429
Iteration 14/30, Best Fitness: 1.5138101828809285
Iteration 15/30, Best Fitness: 1.5138101828809285
Iteration 16/30, Best Fitness: 1.300269684490209
Iteration 17/30, Best Fitness: 1.300269684490209
Iteration 18/30, Best Fitness: 1.300269684490209
Iteration 19/30, Best Fitness: 1.2738498249584989
Iteration 20/30, Best Fitness: 1.1445834652176474
Iteration 21/30, Best Fitness: 0.8487556087655604
Iteration 22/30, Best Fitness: 0.8487556087655604
Iteration 23/30, Best Fitness: 0.8289231635578032
Iteration 24/30, Best Fitness: 0.8242402471719793
Iteration 25/30, Best Fitness: 0.5258270013075049
Iteration 26/30, Best Fitness: 0.5258270013075049
Iteration 27/30, Best Fitness: 0.3996236442626478
Iteration 28/30, Best Fitness: 0.3996236442626478
Iteration 29/30, Best Fitness: 0.3996236442626478
Iteration 30/30, Best Fitness: 0.3996236442626478



Program 5: Grey Wolf Optimizer (GWO)

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of wolves and the number of iterations.
3. Initialize Population: Generate an initial population of wolves with random positions.
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.
5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.
6. Iterate: Repeat the evaluation and position updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations

Algorithm:

28/10/24

Grey Wolf Optimizer

Mangal

Date _____
Page _____

5) Grey wolf Optimizer (GWO):

1) Objective function:

→ Define the function to evaluate the fitness of a given solution x (Sphere function).

→ Function obj_fun(x):

return sum(x^2)

6) Initialize the parameters:

→ Set up the parameters, including the number of wolves, number of dimensions in optimization problem, no. of iterations & the bounds for search space.

→ set num-wolves = 5

set num-dim = 2

set num-iterations = 100

set lb = -10

set up = 10

7) Initialize population:

→ Each wolf's position is initialized randomly within the search space bounds. Each wolf is candidate solution represented by a vector in multi-dimensional space.

→ Function initialize_wolves (num-wolves, num-dim, bounds):

population = []

for i in range (num-wolves):

position = []

for d in range (num-dim):

position[d] = random_value (bounds)

population.append(position)

Galaxy F54 5G return population

4) Evaluate Fitness & assign alpha, beta, delta wolves:

→ Evaluate the fitness of each wolf. Update the alpha, beta & delta wolves based on their fitness scores. The wolf with the lowest fitness score becomes the alpha wolf. The second lowest beta & the third becomes delta

⇒ Function evaluate-fit():

Initialize alpha-pos, beta-pos, delta-pos as zero values.

Initialize alpha-score, beta-score, delta-score as infinity

For each wolf in wolves:

fitness = obj-fun(wolf)

If fitness < alpha-score:

update alpha, beta, delta with the current wolf

else if fitness < beta-score:

update beta, delta with current wolf

else if fitness < delta-score:

update delta with the current wolf

5) Update positions of wolves:

→ Update the position of each wolf using the positions of the alpha, beta & delta wolves. The update equation incorporates random factors to simulate exploration & exploitation. The parameter decreases over time to reduce exploration as the algorithm converges

\Rightarrow Function update-posn (iteration);
 $a = 2 - (\text{iteration} * (2 / \text{num_iterations}))$
 for each wolf in wolves:
 for each dimension j :
 $x_1, x_2 = \text{random values b/w } 0 \text{ & } 1$
 $x_1 = \alpha_{\text{pos}}[j] - (2 * a * \alpha_1 - a) * \text{abs}(2 * \alpha_2 * \alpha_{\text{pos}}[j] - w_{\text{of}}[j])$
 $x_2 = \beta_{\text{pos}}[j] - (2 * a * \alpha_1 - a) * \text{abs}(2 * \alpha_2 * \beta_{\text{pos}}[j] - w_{\text{of}}[j])$
 $x_3 = \delta_{\text{pos}}[j] - (2 * a * \alpha_1 - a) * \text{abs}(2 * \alpha_2 * \delta_{\text{pos}}[j] - w_{\text{of}}[j])$
 $w_{\text{of}}[j] = (x_1 + x_2 + x_3) / 3$

$$w_{\text{of}}[j] = \text{clamp}(w_{\text{of}}[j], \text{lb}, \text{ub})$$

6) Iterate (Position update)

\rightarrow For each iteration, evaluate the fitness of the wolves
 & update their positions. The alpha (best) score is recorded for monitoring the convergence of the algorithm.

\Rightarrow For iteration from 1 to num-iterations:
 call evaluate-fitness()
 call update-positions (iteration)
 record alpha-score in history
 print iteration number & alpha-score

7) Output the Best Solution:
→ After completing all iterations, output the best solution found & its fitness score.

⇒ print "Best solution:", alpha-pos
print "Best solve fitness:", alpha-score.

8) Plot convergence.

→ Plot the alpha score over iterations to visualize how the optimization process converges towards the optimal solution.

⇒ Plot alpha-score history against iteration no
set title to "Convergence of Grey Wolf optimization"
set x-label to "Iteration"
set y-label to "Alpha Fitness score"
Display the plot.

* Applications:-

→ Machine learning (model training, feature selection)
→ Robotics (path planning)

$$S_{\text{total}} \approx 28^{1/24}$$

Code:

```
#GWO
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the Problem (a mathematical function to optimize)
def objective_function(x):
    return np.sum(x**2) # Example: Sphere function (minimize sum of squares)

# Step 2: Initialize Parameters
num_wolves = 5 # Number of wolves in the pack
num_dimensions = 2 # Number of dimensions (for the optimization problem)
num_iterations = 30 # Number of iterations
lb = -10 # Lower bound of search space
ub = 10 # Upper bound of search space

# Step 3: Initialize Population (Generate initial positions randomly)
wolves = np.random.uniform(lb, ub, (num_wolves, num_dimensions))

# Initialize alpha, beta, delta wolves
alpha_pos = np.zeros(num_dimensions)
beta_pos = np.zeros(num_dimensions)
delta_pos = np.zeros(num_dimensions)

alpha_score = float('inf') # Best (alpha) score
beta_score = float('inf') # Second best (beta) score
delta_score = float('inf') # Third best (delta) score

# To store the alpha score over iterations for graphing
alpha_score_history = []

# Step 4: Evaluate Fitness and assign Alpha, Beta, Delta wolves
def evaluate_fitness():
    global alpha_pos, beta_pos, delta_pos, alpha_score, beta_score, delta_score

    for wolf in wolves:
        fitness = objective_function(wolf)

        if fitness < alpha_score:
            alpha_pos = wolf
            alpha_score = fitness
        elif fitness < beta_score:
            beta_pos = wolf
            beta_score = fitness
        elif fitness < delta_score:
            delta_pos = wolf
            delta_score = fitness
```

```

delta_score = beta_score
delta_pos = beta_pos.copy()

beta_score = alpha_score
beta_pos = alpha_pos.copy()

alpha_score = fitness
alpha_pos = wolf.copy()
elif fitness < beta_score:
    delta_score = beta_score
    delta_pos = beta_pos.copy()

beta_score = fitness
beta_pos = wolf.copy()
elif fitness < delta_score:
    delta_score = fitness
    delta_pos = wolf.copy()

# Step 5: Update Positions
def update_positions(iteration):
    a = 2 - iteration * (2 / num_iterations) # a decreases linearly from 2 to 0

    for i in range(num_wolves):
        for j in range(num_dimensions):
            r1 = np.random.random()
            r2 = np.random.random()

            # Position update based on alpha
            A1 = 2 * a * r1 - a
            C1 = 2 * r2
            D_alpha = abs(C1 * alpha_pos[j] - wolves[i, j])
            X1 = alpha_pos[j] - A1 * D_alpha

            # Position update based on beta
            r1 = np.random.random()
            r2 = np.random.random()
            A2 = 2 * a * r1 - a
            C2 = 2 * r2
            D_beta = abs(C2 * beta_pos[j] - wolves[i, j])
            X2 = beta_pos[j] - A2 * D_beta

```

```

# Position update based on delta
r1 = np.random.random()
r2 = np.random.random()
A3 = 2 * a * r1 - a
C3 = 2 * r2
D_delta = abs(C3 * delta_pos[j] - wolves[i, j])
X3 = delta_pos[j] - A3 * D_delta

# Update wolf position
wolves[i, j] = (X1 + X2 + X3) / 3

# Apply boundary constraints
wolves[i, j] = np.clip(wolves[i, j], lb, ub)

# Step 6: Iterate (repeat evaluation and position updating)
for iteration in range(num_iterations):
    evaluate_fitness() # Evaluate fitness of each wolf
    update_positions(iteration) # Update positions based on alpha, beta, delta

    # Record the alpha score for this iteration
    alpha_score_history.append(alpha_score)

    # Optional: Print current best score
    print(f"Iteration {iteration+1}/{num_iterations}, Alpha Score: {alpha_score}")

# Step 7: Output the Best Solution
print("Best Solution:", alpha_pos)
print("Best Solution Fitness:", alpha_score)

# Plotting the convergence graph
plt.plot(alpha_score_history)
plt.title('Convergence of Grey Wolf Optimizer')
plt.xlabel('Iteration')
plt.ylabel('Alpha Fitness Score')
plt.grid(True)
plt.show()

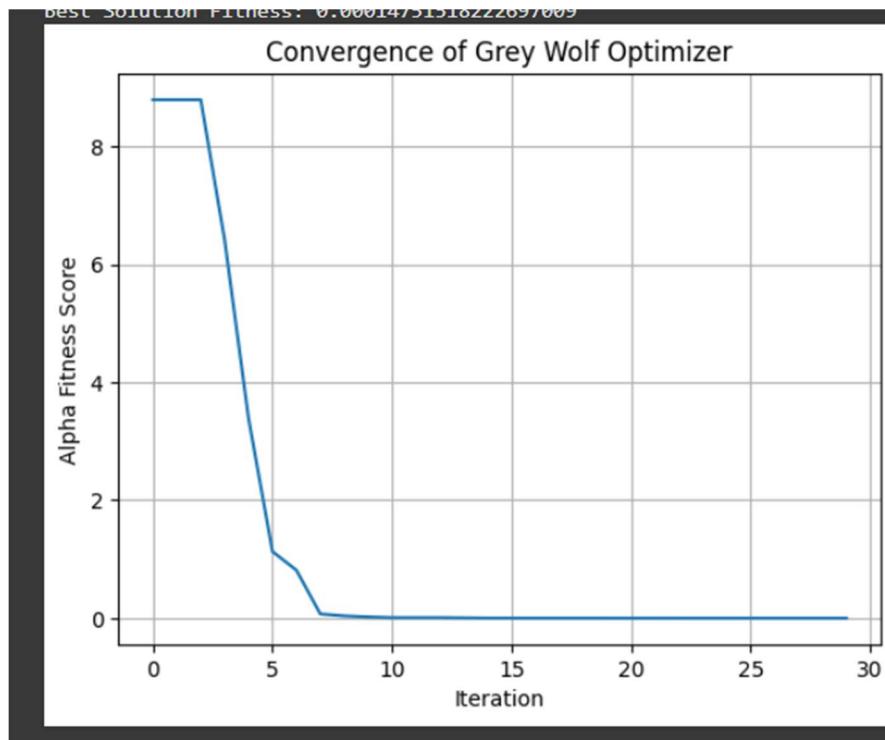
```

OUTPUT:

```

→ Iteration 1/30, Alpha Score: 8.789922247101906
Iteration 2/30, Alpha Score: 8.789922247101906
Iteration 3/30, Alpha Score: 8.789922247101906
Iteration 4/30, Alpha Score: 6.409956649485766
Iteration 5/30, Alpha Score: 3.383929841190778
Iteration 6/30, Alpha Score: 1.1292299489236237
Iteration 7/30, Alpha Score: 0.8136628488047792
Iteration 8/30, Alpha Score: 0.07110881373527288
Iteration 9/30, Alpha Score: 0.03823180120070083
Iteration 10/30, Alpha Score: 0.021111314445105462
Iteration 11/30, Alpha Score: 0.00874782100259989
Iteration 12/30, Alpha Score: 0.00874782100259989
Iteration 13/30, Alpha Score: 0.00874782100259989
Iteration 14/30, Alpha Score: 0.005066807028932165
Iteration 15/30, Alpha Score: 0.0011746187200998674
Iteration 16/30, Alpha Score: 0.0011746187200998674
Iteration 17/30, Alpha Score: 0.0008078646351838173
Iteration 18/30, Alpha Score: 0.0008078646351838173
Iteration 19/30, Alpha Score: 0.0006302256737926024
Iteration 20/30, Alpha Score: 0.0005272190797352655
Iteration 21/30, Alpha Score: 0.00035614966782860404
Iteration 22/30, Alpha Score: 0.0003270119398391142
Iteration 23/30, Alpha Score: 0.00022723766847392013
Iteration 24/30, Alpha Score: 0.00022152382849585967
Iteration 25/30, Alpha Score: 0.00022152382849585967
Iteration 26/30, Alpha Score: 0.00020102313789207912
Iteration 27/30, Alpha Score: 0.0001974565833678501
Iteration 28/30, Alpha Score: 0.0001547675581999543
Iteration 29/30, Alpha Score: 0.00014751518222697009
Iteration 30/30, Alpha Score: 0.00014751518222697009
Best Solution: [ 0.00643925 -0.01029812]
Best Solution Fitness: 0.00014751518222697009

```



Program 6: Parallel Cellular Algorithms and Programs

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.
3. Initialize Population: Generate an initial population of cells with random positions in the solution space.
4. Evaluate Fitness: Evaluate the fitness of each cell based on the optimization function.
5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.
6. Iterate: Repeat the evaluation and state updating process for a fixed number of iterations or until convergence criteria are met.
7. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

8/12/29

Parallel Cellular Algorithm

Mangal
Delete
Page

a) Parallel Cellular Algorithm

or Optimization function:

→ Implements the Sphere function, used to evaluate fitness by summing the square of all position values.

→ Function opt-func (posn):

return sum of squares of all elements in 'position'

b) Initialize population:

→ Function init-pop (grid size, dim):

create population array with random values in range [-10, 10] of size (gridsize[0], gridsize[1], dimensions)

return population

c) Evaluate fitness:

→ Evaluates fitness by applying the optimization function to each cell's position.

→ Function evaluate-fitness (population):

Initialize fitness array of zeroes with the same 2D shape as 'population'

For each cell (i, j) in 'population'

calculate fitness $[i, j] = \text{optfun}(\text{population}[i, j])$

Return fitness

d) Get neighbourhood:

→ Retrieves valid neighbouring cells within a specified radius in the grid.

\Rightarrow Function `get_neighborhood(grid, x, y, radius)`:
 Initialize empty 'neighbors' list
 For each offset (i, j) within the radius:
 If (i, j) is not $(0, 0)$:
 calculate $n_i = x + i, n_j = y + j$
 If n_i and n_j are within grid bounds
 Add (n_i, n_j) to neighbors

Return neighbors.

5) Update States

\Rightarrow Update cell's position by averaging it with its best-performing neighbor's position.

\Rightarrow Function `update_states(population, fitness)`:

Initialize 'new-population' as a copy of 'population'

For each cell (i, j) in the grid:

Get neighbors of cell (i, j) using `get_neighborhood`

Initialize best-neighbor = current cell's position

Initialize best-fitness = fitness of current cell

For each cell neighbor (n_i, n_j) :

If $\text{fitness}[n_i, n_j] < \text{best-fitness}$:

 update best-fitness = $\text{fitness}[n_i, n_j]$

 update best-neighbor = position of neighbor (n_i, n_j)

 update new-pop[i, j] = $(\text{curr-pos} + \text{best-neighbor}) / 2$

return new pop.

6) Main function : Parallel Cellular Algorithm

\Rightarrow The main loop initializes the grid, evaluates

F54 5G fitness, updates positions, tracks the best solution.

→ Function parallel_cellular_algo():

 Initialize pop = init_pop (grid size, dim)

 Initialize best_sol = None

 Initialize best_fitness = infinity

For iteration = 1 to Iterations:

 calculate fitness = eval_fitness (pop)

 Find min fit and corresponding posn from
 fitness array.

 If min_fitness < best_fitness :

 update best_fitness = min_fitness

 update best_solution = corresponding posn

 update pop = update_states (pop, fitness)

 print iteration and best_fitness

print best soln & best fitness.

* Applications

→ Cryptography, ML

```

#pcap
import numpy as np

# Define the problem: A simple optimization function (e.g., Sphere Function)
def optimization_function(position):
    """Example: Sphere Function for minimization."""
    return sum(x**2 for x in position)

# Initialize Parameters
GRID_SIZE = (10, 10) # Grid size (rows, columns)
NEIGHBORHOOD_RADIUS = 1 # Moore neighborhood radius
DIMENSIONS = 2 # Number of dimensions in the solution space
ITERATIONS = 30 # Number of iterations

# Initialize Population
def initialize_population(grid_size, dimensions):
    """Initialize a grid with random positions."""
    population = np.random.uniform(-10, 10, size=(grid_size[0], grid_size[1], dimensions))
    return population

# Evaluate Fitness
def evaluate_fitness(population):
    """Calculate the fitness of all cells."""
    fitness = np.zeros((population.shape[0], population.shape[1]))
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = optimization_function(population[i, j])
    return fitness

# Get Neighborhood
def get_neighborhood(grid, x, y, radius):
    """Get the neighbors of a cell within the specified radius."""
    neighbors = []
    for i in range(-radius, radius + 1):
        for j in range(-radius, radius + 1):
            if i == 0 and j == 0:
                continue # Skip the current cell
            ni, nj = x + i, y + j
            if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
                neighbors.append((ni, nj))
    return neighbors

# Update States
def update_states(population, fitness):
    """Update the state of each cell based on its neighbors."""
    new_population = np.copy(population)

```

```

for i in range(population.shape[0]):
    for j in range(population.shape[1]):
        neighbors = get_neighborhood(population, i, j, NEIGHBORHOOD_RADIUS)
        best_neighbor = population[i, j]
        best_fitness = fitness[i, j]

        # Find the best position among neighbors
        for ni, nj in neighbors:
            if fitness[ni, nj] < best_fitness:
                best_fitness = fitness[ni, nj]
                best_neighbor = population[ni, nj]

        # Update the cell state (move towards the best neighbor)
        new_population[i, j] = (population[i, j] + best_neighbor) / 2 # Average position
    return new_population

# Main Algorithm
def parallel_cellular_algorithm():
    """Implementation of the Parallel Cellular Algorithm."""
    population = initialize_population(GRID_SIZE, DIMENSIONS)
    best_solution = None
    best_fitness = float('inf')

    for iteration in range(ITERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness = np.min(fitness)
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(np.argmin(fitness), fitness.shape)]

        # Update states based on neighbors
        population = update_states(population, fitness)

        # Print progress
        print(f"Iteration {iteration + 1}: Best Fitness = {best_fitness}")

    print("\nBest Solution Found:")
    print(f"Position: {best_solution}, Fitness: {best_fitness}")

# Run the algorithm
if __name__ == "__main__":
    parallel_cellular_algorithm()

```

OUTPUT:

```
➡ Iteration 1: Best Fitness = 0.43918427791098213
➡ Iteration 2: Best Fitness = 0.43918427791098213
➡ Iteration 3: Best Fitness = 0.062221279350329436
Iteration 4: Best Fitness = 0.030149522005462108
Iteration 5: Best Fitness = 0.015791278460696168
Iteration 6: Best Fitness = 0.0025499667118763104
Iteration 7: Best Fitness = 0.0025499667118763104
Iteration 8: Best Fitness = 0.00019007166980743008
Iteration 9: Best Fitness = 0.00019007166980743008
Iteration 10: Best Fitness = 1.0432171933623911e-05
Iteration 11: Best Fitness = 8.406928148912647e-06
Iteration 12: Best Fitness = 5.511032710180021e-07
Iteration 13: Best Fitness = 4.3084388056725156e-07
Iteration 14: Best Fitness = 2.315054420755622e-07
Iteration 15: Best Fitness = 5.245753459404661e-08
Iteration 16: Best Fitness = 5.245753459404661e-08
Iteration 17: Best Fitness = 4.341357920017173e-08
Iteration 18: Best Fitness = 1.145644119860328e-08
Iteration 19: Best Fitness = 3.147791691706415e-09
Iteration 20: Best Fitness = 2.8192306881167533e-09
Iteration 21: Best Fitness = 9.788374665398935e-11
Iteration 22: Best Fitness = 9.788374665398935e-11
Iteration 23: Best Fitness = 9.788374665398935e-11
Iteration 24: Best Fitness = 9.788374665398935e-11
Iteration 25: Best Fitness = 7.537171686605552e-11
Iteration 26: Best Fitness = 7.234639306921671e-11
Iteration 27: Best Fitness = 7.028872029493468e-11
Iteration 28: Best Fitness = 3.340290444524624e-11
Iteration 29: Best Fitness = 1.4953679944431498e-11
Iteration 30: Best Fitness = 1.0817118995466254e-11

Best Solution Found:
Position: [-2.92599538e-06 -1.50188883e-06], Fitness: 1.0817118995466254e-11
```

Program 7: Optimization via Gene Expression Algorithms

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Implementation Steps:

1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.
3. Initialize Population: Generate an initial population of random genetic sequences.
4. Evaluate Fitness: Evaluate the fitness of each genetic sequence based on the optimization function.
5. Selection: Select genetic sequences based on their fitness for reproduction.
6. Crossover: Perform crossover between selected sequences to produce offspring.
7. Mutation: Apply mutation to the offspring to introduce variability.
8. Gene Expression: Translate genetic sequences into functional solutions.
9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.
10. Output the Best Solution: Track and output the best solution found during the iterations.

Algorithm:

18/12/24

Gene Expression Algorithm

Mangal

Date _____

Page _____

58 Gene Expression Algorithm :-

or Optimization function :-

→ Evaluate the fitness of a solution using square function.

→ Function opt (soln):
return sum (x^2 for x in soln)

59 Initialize - population :-

→ Function init-pop (pop_size, genes);

create a 2D array 'population' of size (pop_size, genes)

for each individual in population:

 for each gene in the individual:

 Randomly initialize gene value between

-10 and 10

return population

60 Evaluate fitness :-

→ Calculates the fitness for each solution in population

→ Function eval-fit (pop);

 Initialize an empty list 'fitness'

 for each individual 'solution' in population

 fitness_val = opt (sol)

 Append fitness_val to 'fitness'

 return fitness as an array.

61 select parents :-

→ Selects the best individuals for reproduction using tournament selection.

→ Function select-parents (pop, fitness, num_parents);

 initialize an empty list 'parents'

For i from 1 to num_parents:

Randomly select 3 indices as 'tournament'

Find the index 'best' with minimum fitness among tournament

Append the individual at 'best' index to 'parents'

return parents as an array

e) Crossover :-

→ perform single-point crossover to create offspring.

→ Function crossover (parents, crossover-rate):

Initialize an empty list 'offspring'

For i from 0 to length(parents):

If $i+1$ exceeds length of parents

Break

set parent1 = parents[i]

set parent2 = parents[i+1]

Generate random no. σ

If $\sigma < \text{crossover_rate}$:

Randomly choose a crossover point

'point' (1 to length of genes - 1)

child1 \leftarrow combine genes from parent1

[: point] and parent2 [point :]

child2 \leftarrow combine genes from parent2

[: point] and parent1 [point :]

else:

child1 = parent1

child2 = parent2

Append child1 and child2 to 'offspring'

return offspring as an array

6) Mutate (offspring, m_r):

→ Function mutate (offspring, m_r):

For each individual 'child' in offspring:

For each gene in 'child':

Generate a random m_r'r'

If r < m_r:

modify the gene by adding a small
random value between -1 and 1

return mutated offspring

7) gene-expression algo:

→ Function gene_expression_algo():

set pop = Init_pop (pop-size, genes)

set best_sol = None

set best-fitness = infinity

For gen. from 1 to generations,

fitness = eval_fit (pop)

If fitness [min-fit-idx] < best-fitness:

best-fitness = fitness [min-fit-idx]

best_soln = pop [min-fit-idx]

parents = select_parents (pop, fitness, pop_size)

offspring = crossover (parents, crossover-rate)

offspring = mutate (offspring, mut-rate)

population = gene_expression (offspring)

print best solution and best fitness.

* Application

→ Data Analysis.

✓ RedB
→ 18/12/24

Code:

```
import numpy as np
import random

# 1. Define the Problem: Optimization Function (e.g., Sphere Function)
def optimization_function(solution):
    """Sphere Function for minimization (fitness evaluation)."""
    return sum(x**2 for x in solution)

# 2. Initialize Parameters
POPULATION_SIZE = 50 # Number of genetic sequences (solutions)
GENES = 5 # Number of genes per solution
MUTATION_RATE = 0.1 # Probability of mutation
CROSSOVER_RATE = 0.7 # Probability of crossover
GENERATIONS = 30 # Number of generations to evolve

# 3. Initialize Population
def initialize_population(pop_size, genes):
    """Generate initial population of random genetic sequences."""
    return np.random.uniform(-10, 10, (pop_size, genes))

# 4. Evaluate Fitness
def evaluate_fitness(population):
    """Evaluate the fitness of each genetic sequence."""
    fitness = [optimization_function(solution) for solution in population]
    return np.array(fitness)

# 5. Selection: Tournament Selection
def select_parents(population, fitness, num_parents):
    """Select parents using tournament selection."""
    parents = []
    for _ in range(num_parents):
        tournament = random.sample(range(len(population)), 3) # Randomly select 3 candidates
        best = min(tournament, key=lambda idx: fitness[idx])
        parents.append(population[best])
    return np.array(parents)

# 6. Crossover: Single-Point Crossover
def crossover(parents, crossover_rate):
    """Perform crossover between pairs of parents."""
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 >= len(parents):
            break
        parent1, parent2 = parents[i], parents[i + 1]
        if random.random() < crossover_rate:
```

```

point = random.randint(1, len(parent1) - 1) # Single crossover point
child1 = np.concatenate((parent1[:point], parent2[point:]))
child2 = np.concatenate((parent2[:point], parent1[point:]))
else:
    child1, child2 = parent1, parent2 # No crossover
offspring.extend([child1, child2])
return np.array(offspring)

# 7. Mutation
def mutate(offspring, mutation_rate):
    """Apply mutation to introduce variability."""
    for i in range(len(offspring)):
        for j in range(len(offspring[i])):
            if random.random() < mutation_rate:
                offspring[i][j] += np.random.uniform(-1, 1) # Random small change
    return offspring

# 8. Gene Expression: Functional Solution (No transformation needed for this case)
def gene_expression(population):
    """Translate genetic sequences into functional solutions."""
    return population # Genetic sequences directly represent solutions here.

# 9. Main Function: Gene Expression Algorithm
def gene_expression_algorithm():
    """Implementation of Gene Expression Algorithm for optimization."""
    # Initialize population
    population = initialize_population(POPULATION_SIZE, GENES)
    best_solution = None
    best_fitness = float('inf')

    for generation in range(GENERATIONS):
        # Evaluate fitness
        fitness = evaluate_fitness(population)

        # Track the best solution
        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = population[min_fitness_idx]

        # Selection
        parents = select_parents(population, fitness, POPULATION_SIZE // 2)

        # Crossover
        offspring = crossover(parents, CROSSOVER_RATE)

        # Mutation
        offspring = mutate(offspring, MUTATION_RATE)

        # Evaluate fitness again after mutation
        fitness = evaluate_fitness(offspring)

        # Track the best solution again after mutation
        min_fitness_idx = np.argmin(fitness)
        if fitness[min_fitness_idx] < best_fitness:
            best_fitness = fitness[min_fitness_idx]
            best_solution = offspring[min_fitness_idx]

    return best_solution

```

```

offspring = mutate(offspring, MUTATION_RATE)

# Gene Expression
population = gene_expression(offspring)

# Print progress
print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")

# Output the best solution
print("\nBest Solution Found:")
print(f"Position: {best_solution}, Fitness: {best_fitness}")

# 10. Run the Algorithm
if __name__ == "__main__":
    gene_expression_algorithm()

```

OUTPUT:

```

❯ Generation 1: Best Fitness = 55.82997756903893
Generation 2: Best Fitness = 26.410565738143625
Generation 3: Best Fitness = 21.857647823851615
Generation 4: Best Fitness = 20.016914182036285
Generation 5: Best Fitness = 20.016914182036285
Generation 6: Best Fitness = 20.016914182036285
Generation 7: Best Fitness = 13.81760087982789
Generation 8: Best Fitness = 13.81760087982789
Generation 9: Best Fitness = 12.077725051361178
Generation 10: Best Fitness = 10.461698723345474
Generation 11: Best Fitness = 8.933105023570093
Generation 12: Best Fitness = 6.619449963941974
Generation 13: Best Fitness = 3.1567413435369454
Generation 14: Best Fitness = 3.1567413435369454
Generation 15: Best Fitness = 3.1567413435369454
Generation 16: Best Fitness = 2.74585545305795
Generation 17: Best Fitness = 2.7031453676198964
Generation 18: Best Fitness = 2.078188177116774
Generation 19: Best Fitness = 1.5193087227027497
Generation 20: Best Fitness = 1.4413606561895607
Generation 21: Best Fitness = 0.8501569187378994
Generation 22: Best Fitness = 0.4209372164676112
Generation 23: Best Fitness = 0.3893761873774093
Generation 24: Best Fitness = 0.3893761873774093
Generation 25: Best Fitness = 0.3893761873774093
Generation 26: Best Fitness = 0.3741053651316379
Generation 27: Best Fitness = 0.1381555631914642
Generation 28: Best Fitness = 0.12238160343023853
Generation 29: Best Fitness = 0.12238160343023853
Generation 30: Best Fitness = 0.12238160343023853

Best Solution Found:
Position: [-0.03614343 -0.00257499  0.02260677  0.31412563  0.14792784], Fitness: 0.12238160343023853

```